STREX: Boosting Instruction Cache Reuse in OLTP Workloads Through Stratified Transaction Execution

Islam Atta[†] Pınar Tözün[‡] Xin Tong[†] Anastasia Ailamaki[‡] Andreas Moshovos[†]

[‡]École Polytechnique Fédérale de Lausanne
Lausanne, VD, Switzerland
{pinar.tozun,anastasia.ailamaki}@epfl.ch

[†]University of Toronto
Toronto, ON, Canada
{iatta,xtong,moshovos}@eecg.toronto.edu

ABSTRACT

Online transaction processing (OLTP) workload performance suffers from instruction stalls; the instruction footprint of a typical transaction exceeds by far the capacity of an L1 cache, leading to ongoing cache thrashing. Several proposed techniques remove some instruction stalls in exchange for error-prone instrumentation to the code base, or a sharp increase in the L1-I cache unit area and power. Others reduce instruction miss latency by better utilizing a shared L2 cache. SLICC [2], a recently proposed thread migration technique that exploits transaction instruction locality, is promising for high core counts but performs sub-optimally or may hurt performance when running on few cores.

This paper corroborates that OLTP transactions exhibit significant intra- and inter-thread overlap in their instruction footprint, and analyzes the instruction stall reduction benefits. This paper presents STREX, a hardware, programmer-transparent technique that exploits typical transaction behavior to improve instruction reuse in first level caches. STREX time-multiplexes the execution of similar transactions dynamically on a single core so that instructions fetched by one transaction are reused by all other transactions executing in the system as much as possible. STREX dynamically slices the execution of each transaction into cache-sized segments simply by observing when blocks are brought in the cache and when they are evicted. Experiments show that, when compared to baseline execution on 2 – 16 cores, STREX consistently improves performance while reducing the number of L1 instruction and data misses by 37% and 14% on average, respectively. Finally, this paper proposes a practical hybrid technique that combines STREX and SLICC, thereby guaranteeing performance benefits regardless of the number of available cores and the workload's footprint.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*cache memories*; C.1.4 [**Processor Architectures**]: Parallel Architectures; H.2.4 [**Database Management**]: Systems—*transaction processing, concurrency*

General Terms

Performance, Design.

Keywords

OLTP, instruction cache, instruction locality, thread scheduling

ISCA '13 Tel-Aviv, Israel

1. INTRODUCTION

Online transaction processing (OLTP) is a 100 billion-$/year industry, which is expected to continue growing as online services become even more pervasive. Existing microarchitectures are not tailored well to OLTP execution needs, with instruction stalls accounting for up to 85% of the overall execution cycles [6, 26]. Most OLTP transactions have relatively large active instruction footprints (in excess of 128KB per transaction) that thrash existing first-level instruction caches (L1-I). Several works propose to alleviate instruction stalls using hardware [2, 5, 7, 13, 20] or software [10] techniques. Existing techniques effectively reduce the number of L1-I misses or the associated penalty when running OLTP workloads, but in return either require error-prone instrumentation to the software code base, or employ hardware prefetching tables that more than double the area devoted to the L1-I cache units. Others reduce instruction miss latency by better utilizing the aggregate L2 cache capacity.

Recent work observes that CMP integration offers an alternate path to instruction stall reduction. Specifically, SLICC [2] exploits the following observations: (1) beyond a certain core count, the aggregate L1 capacity of a chip multiprocessor becomes sufficiently large to capture not only the footprint of each transaction individually, but the footprints of all of the concurrently running transactions. (2) There is significant overlap in the instruction code segments executed within and across transactions [10]. SLICC dynamically migrates transaction execution across cores to avoid instruction cache thrashing, allowing multiple transactions to reuse cached instructions. Through thread migration SLICC exploits both intra- and inter-transaction overlap by converting it into instruction reuse in the caches. SLICC was demonstrated on a 16-core CMP, but this work demonstrates that it is not as effective and may *hurt*

performance when the footprint of all concurrently running transactions exceeds the aggregate L1-I capacity.

While main-stream 16-core server CMPs are around the corner and the number of cores on-chip is expected to grow in the future, the number of cores per application may not always be sufficient. The data center design and deployment, and application trends influence the available per application core count. (1) Current data center design trends are toward consolidating more virtual machines on servers as this increases utilization, improves security and energy efficiency, and reduces costs and management overheads [4, 12, 29]. (2) A data center may run multiple OLTP workloads, each with different transactions. (3) While L1-I capacities remain cycle-time limited, OLTP transaction instruction footprints are increasing. Transactions are becoming more complex and thus larger as a result of additional functionality such as data analytics (e.g., WebSphere [11], WebLogic [18]) or more complex logic. Accordingly, it is desirable to avoid SLICC's performance cliff and to develop an instruction stall reduction technique that is effective irrespective of the number of available cores.

This work demonstrates that there is significant temporal code overlap among similar OLTP transactions, suggesting that their execution order can be stratified to increase instruction reuse in the caches. Motivated by this observation, this work proposes STREX, a technique that exploits inter-transaction locality by grouping and synchronizing the execution of similar transactions into time slices. During each time slice, a *lead* transaction brings into the L1-I an instruction code segment that all other transactions ought to reuse. Ideally, when the transactions within a group overlap perfectly, only the lead transaction incurs all necessary misses; these are the misses that a transaction would incur on a conventional system anyhow. As a result of STREX's time slicing, the remaining transactions find all the instructions they need in the L1-I avoiding misses completely. STREX uses local information at each core observing cache block allocations and evictions to orchestrate transaction execution. STREX substantially reduces instruction stall time when running OLTP workloads, while remaining effective even when the aggregate L1-I capacity is insufficient to store the footprints of all concurrently executing transactions. STREX's approach to reducing instruction misses is similar in spirit to STEPS [10]. STEPS is a software technique that was demonstrated on single cores. STEPS relies on manual code instrumentation, a high-overhead error prone process, and produces platform dependent code that is not portable. STREX does not suffer from these limitations as it is a programmer transparent technique.

Experiments demonstrate that while STREX performs substantially better than SLICC when the number of cores is limited, the opposite is true when there are enough cores available. Accordingly, this work proposes a mechanism for dynamically selecting between the two techniques. The selection mechanism uses the available core count while periodically and transparently sampling transactions to measure their footprint needs.

The analysis of inter-transaction overlap shows that for 16 similar transactions concurrently executing on 16 cores, more than 70% of the instruction blocks read during an interval appear in the instruction caches of at least five other cores, while the code overlap is even higher most of the time. This work compares STREX with a baseline conventional CMP architecture, with PIF (the state-of-the-art instruction prefetcher), and with SLICC. When compared to the baseline, STREX exploits transaction code overlap and reduces L1 instruction and data misses respectively by 37% and 13%, on average for two to 16 cores. When compared to the best instruction prefetcher known to date, PIF [7], STREX's performance is within 5% for one workload and 9% better for another. If the core count is so limited that the available aggregate on-chip capacity does not fit the transactions' instruction footprint, STREX outperforms SLICC by an average of 49%. The experiments show that the hybrid mechanism closely follows the performance of the best-performing technique (SLICC or STREX) given the number of cores available and the workload's footprint. Finally, the experiments show that STREX is more effective at reducing instruction misses compared to replacement policies that improve over LRU.

The rest of the paper is organized as follows. Section 2 motivates STREX by discussing the code structure of OLTP transactions and by analyzing the inter-transaction temporal instruction overlap. Section 3 discusses how transaction behavior can be potentially exploited to improve instruction reuse in caches. Section 3 describes STREX and its implementation. Section 5 demonstrates experimentally the performance benefits of STREX. Section 6 reviews related work, while Section 7 concludes.

2. MOTIVATION

Conventional OLTP systems make no explicit effort to improve instruction reuse in processor caches. OLTP systems typically assign transactions to cores in an ad-hoc manner, aiming to balance the work across nodes. A transaction is assigned to a core where it executes to completion. As a result, as the number and the length of transactions in the system increases, OLTP workloads suffer from instruction stalls due to high L1-I cache miss rates. This section demonstrates that OLTP transactions exhibit substantial code overlap, which can create significant locality across different instances of transactions of the same type. Therefore, there is opportunity to improve instruction reuse by coordinating transaction execution.

Section 2.1 takes a closer look at the code structure of transactions revealing that same-type transactions ought to naturally exhibit significant temporal locality. Section 2.2 shows that same-type transactions do follow similar execution paths touching mostly overlapping code segments. STREX relies on this inter-transaction locality to improve instruction reuse by slicing and synchronizing the execution of multiple transactions of the same type over the L1-I of a single core.

The rest of this section focuses on the TPC-C [27] `New Order` and `Payment` transactions, which comprise 88% of the TPC-C workload mix, according to the benchmark's specification. The discussion and results for the other transactions are similar, so they are omitted for brevity. The observations about code structure and instruction code segment overlap apply to all TPC-C and TPC-E [28] transactions.

2.1 OLTP Transaction Code Structure

This section discusses the code structure of some representative OLTP transactions. The results of this section demonstrate that at a high-level, transactions of the same

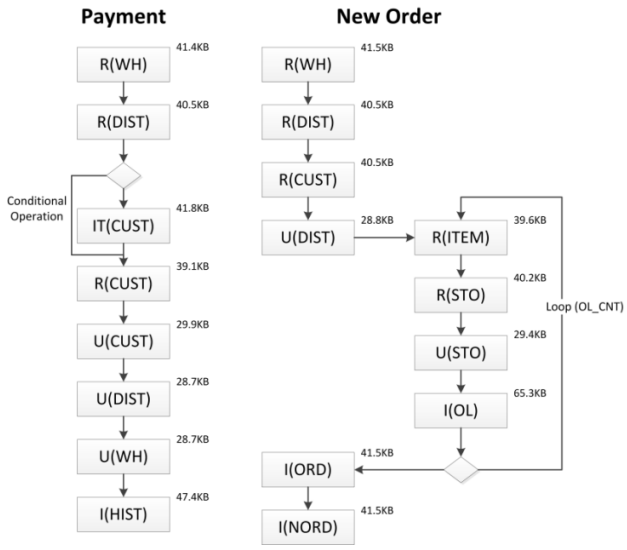

Figure 1: Transaction Flow Graphs tagged with instruction footprint; R: Lookup, U: Update, I: Insert, IT: Index Scan.

type ought to follow similar instruction paths during execution touching similar code segments.

OLTP transactions are composed of *actions* that in turn may execute several basic functions. Basic function examples include looking up a record through an index, scanning and updating an index, inserting a tuple to a table, updating a tuple, etc. No matter how different the output or high-level function of one transaction is from another, all database transactions are composed of a subset of the aforementioned basic functions, repeated several times for different inputs, and various permutations.

Figure 1 shows the action flow graphs for the `New Order` and `Payment` transactions, which represent ∼88% of the transaction mix of TPC-C. Boxes represent *actions*, which are tagged with their corresponding instruction cache footprint; the arrows show the action execution order; and rhombuses represent control flow decisions. The flow diagrams mark each action with the basic function it performs, e.g., R() is an index lookup. Each function manipulates input data in a different way; further details are immaterial to this discussion.

The diagrams suggest that there should be significant overlap across transactions of the same type. All `Payment` transactions execute the same sequence of actions, with the exception of conditionally executing the `IT(CUST)` action. The instruction stream may not be identical across all `Payment` transactions, as the actual code path may vary depending on the input data. Nevertheless, there is overlap for data-independent code segments, and for data-dependent code segments that follow the same path according to similar control-flow decisions. `New Order` transactions exhibit significant overlap as well, despite that their code paths may diverge more over time due to the inner loop conditioned on the `OL_CNT` value.

The diagrams also suggest that there is overlap across transactions of different types. Initially, `New Order` and `Payment` transactions perform index lookups on the same tables: `Warehouse`, `District`, `Customer`. Therefore, their code paths are similar at first, and then they diverge. `New Order` has a loop that executes other statements, while `Payment` updates the previously searched tuples and inserts a tuple to `History` table. Even so, there is overlap as the actions do call common basic functions, albeit with different data.

2.2 Inter-Transaction Temporal Overlap

Figure 2 depicts the degree of instruction footprint overlap over time for `New Order` and `Payment` transactions executing concurrently. This experiment uses 16 randomly chosen same-type transactions that concurrently execute on 16 cores, each with a 32KB L1-I at a rate of one instruction per cycle. Every 100 instructions per core, the *instruction block overlap* is measured for the unique instruction blocks that were touched per core during this interval. The instruction overlap for a block is the number of L1-I caches that contain this instruction block. The graphs show the range of overlap (1, < 5, < 10, and ≥ 10). The measurements stop when at least half of the threads complete execution.

The results show that more than 70% of the instruction blocks touched during an interval appear in at least five other cores. The overlap is even higher most of the time and more than 40% of the instructions touched during an interval appear in at least ten cores. Consequently, *most of the instructions read by a given transaction are also read by at least five other transactions, and about 40% or more of these instructions are also read by at least ten transactions.* In addition, all overlapping instruction fetches happen close enough in time as the instructions survive long enough to be detected in other caches. The measurements do show that the transactions diverge, but very few instruction blocks (less than 10%) appear to be read by a single transaction.

The findings of this section serve as motivation for STREX as they show that there is significant temporal locality across transactions of the same type. The next section discusses how this locality can be potentially exploited to improve instruction reuse in the L1-I.

3. EXPLOITING TRANSACTION INSTRUCTION OVERLAP

Given that OLTP transactions exhibit significant instruction footprint overlap over time, their execution order can be stratified to increase instruction reuse in the caches. This section presents two different ways of exploiting *stratified* execution to improve instruction cache reuse. Both methods break transaction execution into slices, where during each slice, a transaction executes through an L1-I sized code segment. The first method that is the basis for STREX, time-multiplexes and synchronizes the execution of slices from multiple transactions over the same core. The second, previously proposed method, SLICC [2], does so across multiple cores using thread migration. Both methods increase instruction reuse by having slices that touch the same code segment execute in sequence over the same cache where that code segment resides.

Conventional Execution: Figure 3(a) shows how three perfectly overlapping transactions would execute under a conventional OLTP system. The example transactions execute three code segments A, B, and C in order. Each segment fits in the L1-I, but any two segments exceed its capacity. When these transactions execute in a conventional OLTP, they take turns thrashing the cache since each ex-

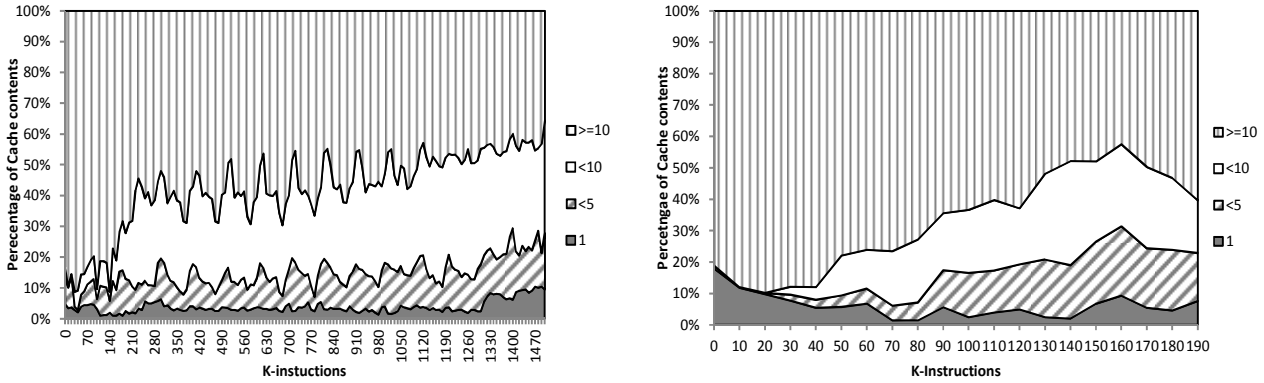

Figure 2: Temporal Overlap Analysis for transactions of type `New Order` (left) and `Payment` (right).

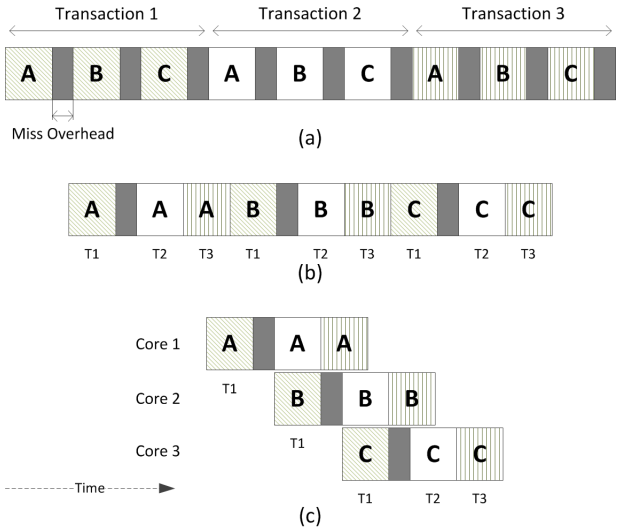

Figure 3: Scheduling Three Identical Transactions. (a) Conventional Execution. (b) STREX. (c) SLICC.

ecutes segments A through C in order before allowing the next transaction to execute. Each segment incurs an overhead due to instruction cache misses. Since the transaction footprint does not fit in the cache, all segments incur an overhead due to misses.

STREX: Figure 3(b) shows a better way of utilizing the L1-I where the first, *lead* transaction executes segment A incurring an overhead as previously. However, instead of proceeding to execute segment B, transaction 1 context switches allowing in turn, transactions 2 and 3 to execute instead. Transactions 2 and 3 find segment A in the L1-I and thus incur no overhead due to misses. Once all three transactions execute the first segment, execution proceeds to segment B and so on. Overall, transaction throughput increases because there are fewer L1-I misses. STREX attempts to achieve the execution order depicted in Figure 3(b) automatically.

SLICC: CMPs enable the migration-based approach used by SLICC. As long as there are enough cores so that the aggregate L1-I capacity can hold all code segments, a transaction can migrate to the core whose L1-I cache holds the code segment the transaction is about to execute. For example, as Figure 3(c) shows, the *lead* transaction can execute segment A first on core 1, then migrate to core 2 where it would execute segment B, then migrate to core 3 where it would execute segment C. Transactions 2 and 3 can follow in a pipelined fashion, finding segments A, B, and C, in cores 1, 2, and 3, respectively. While transaction 1 incurs an overhead when fetching the segments for the first time, the other transactions do not.

STREX and SLICC exploit the same inter-transaction temporal overlap. SLICC has the additional advantage that it can exploit intra-transaction far-flung locality. For example, if the three transactions in Figure 3 executed A, B, and C in a loop, SLICC would only fetch each segment once storing them over three separate L1-Is. STREX would necessarily fetch each segment once per iteration as the segments cannot fit into a single L1-I. However, Section 5.3 shows that SLICC performs well only when there are enough cores to fit the footprint of all concurrently running transactions. In addition, when there are fewer cores in the system, SLICC may even cause performance degradation. The rest of this work investigates a practical implementation for STREX, which can be used when the number of available cores is insufficient for the thread-migration-based SLICC to work well. A system can also decide which method to use while exploiting the best of both worlds. Section 5.5 presents a mechanism that dynamically selects between SLICC and STREX.

4. STREX

The key to the success of STREX is the ability to dynamically detect the points at which a transaction is ought to be context-switched in order to keep inter-transaction execution synchronized, thereby maximizing instruction cache reuse. If a transaction executes for a long time, it will end up evicting cache blocks that other transactions could have reused. If a transaction executes for a short time, the overheads of context switching and of a potential increase in contention in the data caches will overwhelm performance. For these reasons, context switching at regular intervals would perform sub-optimally at best. An optimal *synchronization algorithm* has to rely on dynamic information: a transaction should be allowed to execute as long as it benefits from data and instruction locality, however, it should not be allowed to evict any blocks that will be useful for other transactions.

Moreover, care must be taken to amortize the costs of context switching over the benefits gained as a result of the increase in instruction reuse.

Figure 2 demonstrated significant temporal overlap among same-type transactions, but also showed divergence that results from loops and conditional statements. Thus, breaking down the instruction footprint of several transactions into smaller chunks will not generally result in identical code segments. Optimally scheduling those chunks in order to maximize instruction locality is akin to job scheduling, an NP-complete problem [9]. However, an *optimal* algorithm exists for the simpler case of a group of identical transactions, i.e., transactions that follow the exact same execution path. STREX applies this algorithm to the general case of partially overlapping transactions resulting in a simple, inexpensive solution that performs well.

The rest of this section is organized as follows: Section 4.1 describes and demonstrates the potential of the optimal algorithm for the special case of identical transactions. Section 4.2 presents STREX's core synchronization algorithm while Section 4.3 presents STREX's implementation. Section 4.4 discusses practical challenges and qualitatively compares STREX to SLICC [2] and prefetching.

4.1 Optimal Synchronization for Identical Transactions

For the special case of multiple, perfectly overlapping transactions, there is an algorithm for detecting the optimal context switching points. Let us consider the case of two identical transactions T1 and T2, and let us assume that initially the L1-I is empty. T1 starts executing fetching instruction blocks. Since the transactions are identical, all blocks fetched by T1 will be fetched by T2 in the exact same sequence. T1 should continue execution up to the point where it would be forced to evict a cache block it brought in. Evicting this block would force T2 to refetch it. Once T1 stops, T2 starts executing touching exactly the same blocks that T1 did. All these blocks will be in the L1-I. Once T2 arrives at the point where it would be forced to evict a block, it marks all blocks presently cached with a *phase* number, say 0. T2 continues execution but it marks any block it currently touches with $phase(1)$. T2 continues execution up to the point where it would be forced to evict a block marked as $phase(1)$. Evicting such a block would force T1 to refetch it. Execution continues with T1, up to the point where T1 would now have to evict a block tagged as $phase(1)$. T1 then proceeds to fetch additional blocks, marking them with $phase(2)$. This process continues, incrementing the phase number every time both threads have had a chance to execute with the same phase number.

More formally, the synchronization algorithm for N multiple, identical transactions is as follows: (1) the transactions execute in phases, starting with $phase(0)$. The first transaction is deemed the *lead*. (2) At $phase(i)$, all cache blocks fetched by the *lead* thread are tagged with "i". (3) A transaction, including the *lead*, executes $phase(i)$ as long as it does not evict cache blocks tagged with "i". (4) When the last thread completes $phase(i)$, execution proceeds to $phase(i+1)$.

4.1.1 Potential for Identical Transactions:

As Figure 4 shows, the synchronization algorithm has the potential to work well in practice. The figure shows the reduction in I-MPKI for the L1-I for TPC-C and TPC-E transactions. This experiment uses ten randomly chosen instances for each transaction type. Each of these instances is replicated ten times thus forming a hypothetical workload of 100 transactions. Figure 4 depicts the instruction misses with the optimal synchronization algorithm for each transaction type compared to that of the baseline. The results demonstrate that STREX reduces I-MPKI significantly in all cases.

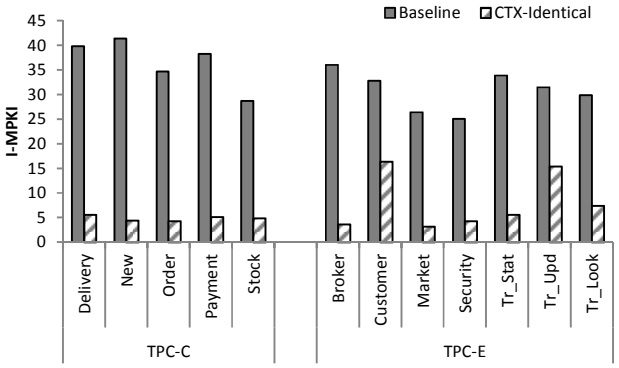

Figure 4: I-MPKI reduction with STREX and identical transactions

However, in practice and as the results of Section 2.2 suggest, rarely different instances of same-type transactions have completely identical instruction streams due to data dependencies. Accordingly, the results of Figure 4 only serve as an indication of what may be possible.

4.2 STREX Synchronization Algorithm

This section presents the synchronization algorithm STREX uses to improve instruction cache reuse for the general case of multiple, non-identical transactions. Since transactions diverge at runtime (as Figure 2 shows), the *lead* may not touch all blocks that the other, subsequent threads need. Hence *non-lead* transactions should also be allowed to fetch new cache blocks.

The generalized STREX synchronization algorithm is as follows:

1. Given a pool of transactions, STREX groups transactions of the same type into *teams*. STREX places each team into a hardware thread queue in an available execution core. Then, it flags the first transaction in the queue as the *lead*.

2. STREX synchronizes transaction execution using a per core $phase_{ID}$ counter. As a transaction touches instruction blocks it tags the block with the current $phase_{ID}$ value no matter whether the access was a hit or a miss. Whenever the *lead* resumes execution, it increments the $phase_{ID}$ counter.

3. STREX continuously monitors *victim* cache blocks. Upon encountering a victim block tagged with the current $phase_{ID}$ value, STREX context switches the current executing transaction and places it at the end of the thread queue. The next ready transaction resumes execution.

4. If the *lead* transaction terminates, the next thread in the queue becomes the *lead*.

5. Threads keep running in a round robin order until they all complete execution.

6. Once all the threads in a team complete execution, the core becomes available for another team to execute.

4.3 Implementation

STREX's implementation requires the following components per core: (a) thread execution queue, (b) a $phase_{ID}$ counter, (c) a $phase_{ID}$ tag per L1-I cache block, (d) a victim block monitoring unit, (e) a thread context switching unit, and (f) STREX's control logic.

STREX tags all cache blocks with $phase_{ID}$ values. These $phase_{ID}$s can be maintained separately in a table (PIDT) to avoid impacting the L1-I design and latency. The *PIDT* contains a $phase_{ID}$ entry per cache block and is accessed in parallel with the L1 tag and data arrays. This work uses 8-bit $phase_{ID}$ tags and an 8-bit, modulo $phase_{ID}$ counter per core. The area overhead of the *PIDT* is small as it uses only eight additional bits per cache block. A PIDT does not contain any address tags or any additional block related information.

This work sizes the scheduling structures empirically in order for STREX to find a sufficient number of similar transactions to improve throughput. The OLTP system can provide up to 30 transactions at any given point in time. STREX groups similar transactions by examining the address of the header instructions similar to SLICC-Pp [2]. STREX groups similar transactions into teams. The maximum number of transactions that a team can have ($team\_size$) is fixed system-wide. STREX assigns teams in the arrival order of the oldest thread in a team. When transactions that are not part of a team (*stray* transactions) become the oldest, they are scheduled individually.

Section 5.4 shows that by controlling the maximum allowed team size, STREX can trade-off between overall throughput and per transaction latency. Software database management scheduling schemes that batch transactions exhibit a similar tradeoff [10, 23, 30].

STREX context switches threads by saving and restoring their architectural state to/from the L2 cache slice nearest to the core. STREX requires support for hardware scheduling of multiple threads. Several proposals exist for implementing hardware-level thread scheduling and context switching (e.g., [22]). STREX serves as additional motivation for further investigating how hardware-level thread scheduling ought to be supported.

4.4 Discussion

This section discusses some of the implications of STREX for corner cases, its overheads, and contrasts it against state-of-the-art instruction miss reduction techniques.

4.4.1 Forward Progress Guarantees

STREX's effectiveness is limited by the amount of temporal overlap available across transactions of the same team. More precisely, the *lead* transaction has the largest impact on locality. For example, in a scenario where the *lead* transaction has minimal temporal overlap with the rest of the team, only the *lead* thread will make forward progress, while the others will have to wait until the *lead* finishes. There is no possibility of deadlock or starvation as the *lead* is guaranteed to finish, and in the worst possible scenario, the rest of the threads will become *leads* in order. Since the *lead* always starts execution with a new $phase_{ID}$, it has the highest authority to evict cache blocks. If other threads do not touch these blocks and try to evict them, they will be context switched too early. Since STREX selects the *lead* randomly, that scenario is unavoidable. Yet, with our examined workloads, this scenario has never happened due to the inherent temporal overlap across transactions of the same type. An extension to STREX might investigate placing lower limits on the amount of forward progress a thread should make before context switching.

4.4.2 Context Switching Overhead

STREX incurs an overhead for context switching among team members. The architectural state of each transaction has to be saved and restored. In this work, thread contexts are saved in the second level cache to avoid thrashing the L1-D. STREX amortizes this overhead by improving instruction and data locality, which result in overall throughput improvement (see Section 5.3). A portion of the physical address space is reserved for storing thread contexts. For the workloads studied, context switches were sufficiently infrequent enough so that the overhead of context saving and restoring was never a significant fraction of the overall execution time. An implementation may choose to enforce a minimum number of instructions or cycles that a transaction ought to execute before a context switch is allowed.

4.4.3 Interaction with Prefetching

Prefetching reduces the latency observed for instruction cache misses by anticipating future instruction fetch requests. The simplest *next-line* prefetcher works well for streaming code regions. Other more sophisticated prefetchers can predict complex execution paths by recording previously seen instruction streams, and then prefetching cache blocks when a part of a recorded stream is touched again. Ferdman *et al.* propose prefetchers that record temporal streams of accessed instruction blocks, *TIFS* [8], or streams of committed instructions, *PIF* [7]. PIF is the most accurate instruction prefetcher known to date for OLTP, but it requires resources that exceed that of a typical L1-I, e.g., ∼40KB per core. When compared to PIF, STREX is expected to require less bandwidth, area, and power. STREX improves locality by increasing instruction and data reuse, rather than by redundantly fetching cache blocks from the L2 at every transaction. In addition, as Section 5.6 shows, STREX uses less than 2% of the storage required by PIF. Finally, STREX does not incur the static and dynamic power for large storage tables and extra bandwidth. STREX, however, is limited to classes of applications that share similar behavior with OLTP workloads, while PIF is a generic technique.

STREX and PIF are not two exclusive solutions to the same problem. STREX can avoid many of the misses that PIF has to incur thus possibly reducing the storage, power, and bandwidth overheads of PIF. PIF could reduce execution time for the *lead* transaction, thus improving performance when used in conjunction with STREX. An investigation of a possible combination of the two techniques is left for future work.

4.4.4 Interaction with SMT

Simultaneous Multithreading (SMT) improves transaction throughput by executing multiple transactions

Table 1: Workloads.

TPC-C-1	1 warehouse, 84 MB Wholesale supplier
TPC-C-10	10 warehouses, 1 GB Wholesale supplier
TPC-E	1000 customers, 20 GB Brokerage house
MapReduce	Hadoop 0.20.2, Mahout 0.4 library Wikipedia page articles (12 GB)

Table 2: System Parameters.

Processing Cores	*N* OoO cores, 2.5GHz 6-wide Fetch/Decode/Issue 128-entry ROB, 80-entry LSQ BTAC (4-way, 512-entry) TAGE (5-tables, 512-entry, 2K-bimod)
Private L1 Caches	32KB, 64B blocks, 8-way 3-cycle load-to-use, 32 MSHRs MESI-coherence for L1-D
L2 NUCA Cache	Shared, 1MB per core, 16-way 64B blocks, *N* slices 16-cycle hit latency, 64 MSHRs
Interconnect	2D Torus, 1-cycle hop latency
Memory	DDR3 1.6GHz, 800MHz Bus, 42ns latency 2 Channels / 1 Rank / 8 Banks 8B Bus Width, Open Page Policy t_{CAS}-10, t_{RCD}-10, t_{RP}-10, t_{RAS}-35 t_{RC}47.5, t_{WR}-15, t_{WTR}-7.5 t_{RTRS}-1, t_{CCD}-4, t_{CWD}-9.5

concurrently over the same core. SMT has been shown to improve performance for OLTP workloads but the expense of additional L1 misses; on real hardware, 2-way SMT increases instruction (15% TPC-C/7% TPC-E) and data (TPC-C 10%/TPC-E 16%) misses [6, 26]. Moreover, the core remains idle 80% of the time suggesting that most stalls remain. It may be possible to use STREX to synchronize thread execution under SMT and thus improve locality and performance. A detailed study of the interaction of STREX and SMT is left for future work.

5. EVALUATION

This section demonstrates experimentally that STREX reduces instruction stalls improving performance for OLTP workloads. Specifically, Section 5.2 demonstrates STREX's impact on instruction and data misses as compared to SLICC [2]. Section 5.3 shows the throughput improvement with STREX in comparison with a next-line prefetcher [24], a state-of-the-art instruction prefetcher (PIF [7]), and SLICC. Section 5.4 highlights that STREX can be tuned to trade-off transaction latency for higher overall throughput. Section 5.5 presents and evaluates a solution combining STREX and SLICC. Section 5.6 discusses the hardware cost of STREX. Section 5.7 investigates the effectiveness of state-of-the-art cache replacement policies, and their interaction with STREX.

5.1 Methodology

Table 1 lists the workloads used. Both TPC-C variants [27] and TPC-E [28] run on top of the scalable open-source storage manager Shore-MT [14], which was shown to have similar components and micro-architectural behavior as commercial DBMSs [1]. The client-driver and the database are kept on the same machine and the buffer-pool is configured to keep the whole database in memory. The experiments use a 1.2B instruction sample from these workloads. TPC-C-1 and TPC-C-10 use two different scaling factors for the TPC-C database and serve to demonstrate that STREX remains effective even when the database size, and thus the data footprint grows larger. TPC-C and TPC-E have larger instruction and data footprints compared to other scale-out workloads [6]. The MapReduce CloudSuite workload [19], which has a relatively small instruction footprint [6] serves to demonstrate that STREX is robust in that it does not reduce performance for workloads that do not have similar behavior to OLTP. The MapReduce workload divides the input dataset across 300 threads, each performing a single map/reduce task. For clarity, the discussion focuses on TPC-C and TPC-E with MapReduce being included only where absolutely necessary.

Conventional operating systems lack support for thread context switching or migration at the hardware level. To work around this limitation, the experiments replay x86 execution traces, modeling the timing of all events, and maintaining the original thread sequence. The traces for TPC-C and TPC-E include both *user* and *kernel* activity, and were collected using QTrace [25], an instrumentation extension to the QEMU full-system emulator [3]. For MapReduce, PIN [17] was used to extract execution traces. All simulations use a modified version of the Zesto x86 multi-core architecture simulator [16].

Table 2 details the baseline architecture. With *N* cores, the baseline architecture has *N* hardware contexts with the OS making thread scheduling decisions. STREX or SLICC form teams over a pool of up to 30 virtual contexts. Unless otherwise noted, each core maintains a thread queue of up to ten threads. STREX forms teams of up to ten threads whereas SLICC forms teams of up to *2N* threads. Throughput is measured as the inverse of the number of cycles required to execute all transactions. The experiments report the misses per kilo instructions for instruction (I-MPKI) and data (D-MPKI).

5.2 L1 Miss Rate

This section shows that STREX improves instruction and data locality consistently for OLTP workloads irrespective of the number of cores. Figure 5 reports the L1 I-MPKI and D-MPKI incurred by the baseline system, SLICC, and STREX for two to 16 cores. For MapReduce, the I- and D-MPKI with STREX is within 1% of the baseline as context switches rarely occur for this workload. The next section shows that performance is virtually identical as well.

The I-MPKI of the baseline is practically independent of the number of cores. This is expected as the baseline makes no explicit effort to increase instruction reuse across threads; adding more cores improves throughput through increased concurrency. STREX consistently reduces I-MPKI over the baseline with the I-MPKI remaining practically constant (the variation is less than 2%) no matter how many cores are available. STREX reduces I-MPKI compared to the baseline

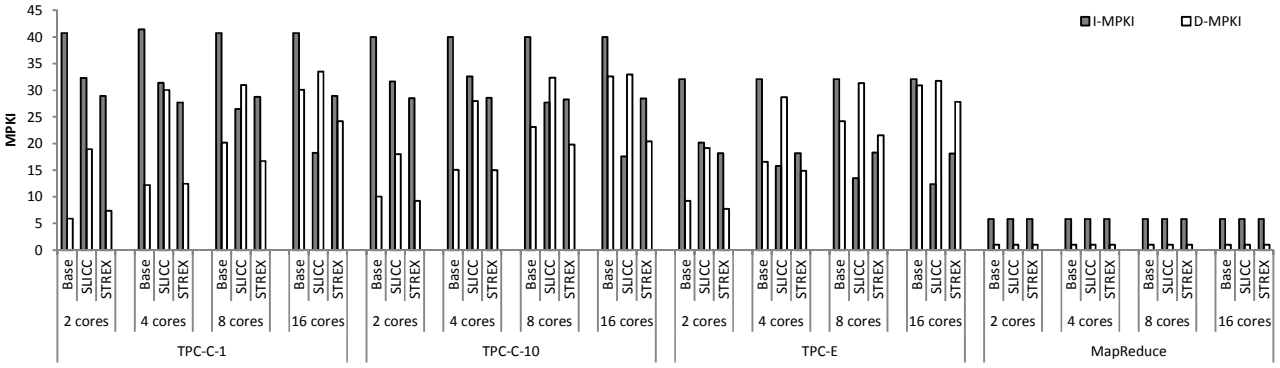

Figure 5: Effect of STREX on L1 instruction and data misses.

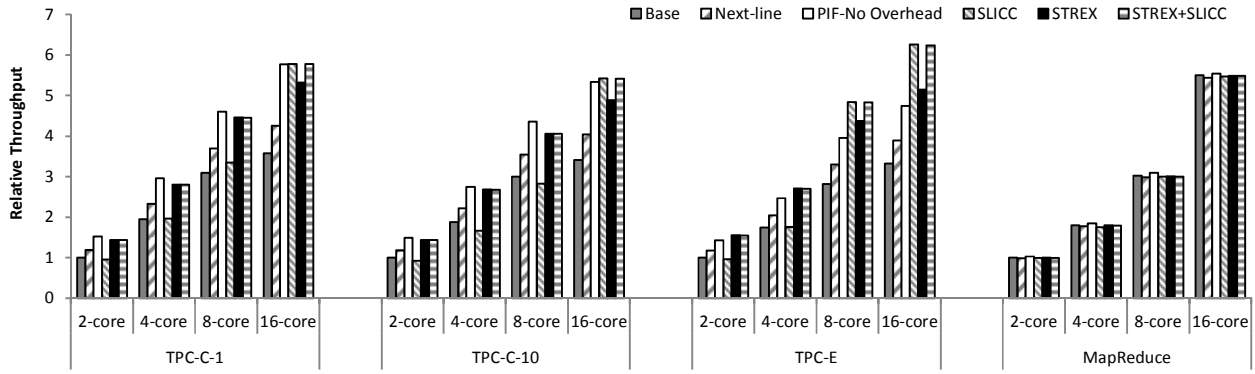

Figure 6: Relative throughput of STREX compared to state-of-the-art instruction miss reduction techniques.

by an average of 30%, 29%, and 44% for TPC-C-1, TPC-C-10, and TPC-E, respectively, and mostly independently of the number of cores.

Figure 5 shows that for the baseline, data misses increase with the number of cores; more concurrency increases coherence misses, since transactions share data. Much of the sharing is due to same type transactions. They tend to access the same metadata and locks of the same tables, as well as the same index roots during index probes, and they tend to do so in the same sequence. STREX improves data locality by synchronizing their execution. The higher the data miss rate of the baseline, the greater the D-MPKI improvements with STREX. For 16 cores, STREX reduces D-MPKI by 20%, 37%, and 11%, for TPC-C-1, TPC-C-10, and TPC-E, respectively.

SLICC behaves differently than STREX improving instructions misses more, as more cores become available. When there are at most eight or four cores for TPC-C and TPC-E, respectively, SLICC does not reduce instruction misses more than STREX. Moreover, data misses always increase with SLICC. Accordingly, even when SLICC matches or improves upon instruction misses compared to STREX, performance may suffer due to increased data misses. As the next section shows, SLICC does not improve and even hurts performance unless there is a sufficient number of cores available.

5.3 Throughput

This section compares the overall throughput of STREX relative to the baseline, a next-line instruction prefetcher [24], a state-of-the-art instruction prefetcher (PIF [7]), and SLICC. Figure 6 reports overall throughput normalized over the 2-core baseline. STREX consistently improves throughput over the baseline by an average of 35–55%, and by 20–32% over the next-line prefetcher, for 2–16 cores. Contrary to SLICC, STREX is insensitive to the number of cores and always improves performance.

SLICC either degrades or barely improves performance over the baseline for the TPC-C workloads with up to eight cores and for the TPC-E workloads with up to four cores. For the same configurations the next-line prefetcher consistently outperforms SLICC. SLICC outperforms STREX and does so considerably only when there are at least eight and 16 cores respectively for TPC-E and for the TPC-C workloads. With 16 cores, SLICC outperforms STREX by 8%, 11% and 21% for TPC-C-1, TPC-C-10 and TPC-E. This result serves as motivation for Section 5.5 which presents and evaluates a hybrid system that combines STREX and SLICC. The system selects dynamically which method to use depending on the number of available cores and the cache demands of the target OLTP workload.

PIF [7] is a state-of-the-art instruction prefetcher that has nearly perfect instruction coverage. The results of Figure 6 are an upper bound for PIF's performance as the experiment models PIF with a 100% hit rate L1-I cache. Demand traffic is generated for cache blocks that would have otherwise missed on a real cache, thus partially modeling the contention that PIF would incur. This is an optimistic 100% accurate prefetcher that issues perfectly timely requests. The actual PIF prefetcher may fail to prefetch in

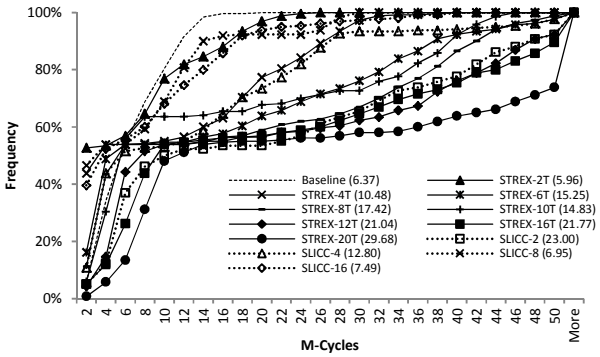

Figure 7: TPC-C transaction latency distribution as a function of *team_size* for STREX, and of core count for SLICC.

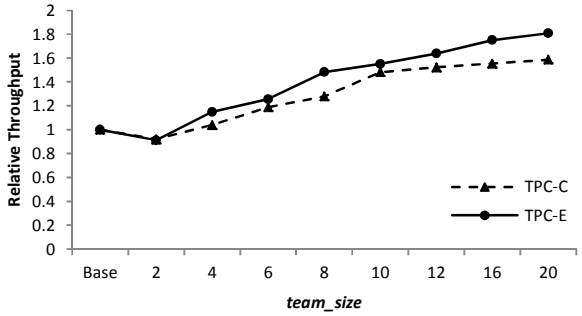

Figure 8: Overall throughput for a range of *team_size* values.

some cases, over-prefetech in others, and not always manage to completely hide the miss latency. For 2–16 cores, STREX achieves on average 95% of PIF's performance for TPC-C, and outperforms PIF by 9% for TPC-E, with less than 2% of the overhead storage.

MapReduce, which has an instruction footprint that fits in the L1-I cache, remains unaffected with all techniques.

5.4 Transaction Throughput vs. Latency

Similar to software transaction batching schemes [10, 23, 30], STREX improves the overall throughput but may increase transaction latency. By adjusting the maximum number of transactions per team (*team_size)*, it is possible to control this trade-off as Figure 7 suggests. Figure 7 shows the distribution of transaction latencies for TPC-C-10 for the baseline, STREX, and SLICC. For STREX the figure reports latency distributions as a function of *team_size*, noted as STREX-xT, where x is a team size in the range of two to 20. In all preceding experiments teams had up to ten threads. With STREX, the transaction latency is independent of the core count, hence the latencies are almost identical and Figure 7 shows latencies for 16 core only. For SLICC the figure reports latency distributions as a function of core count, noted as SLICC-x where x is a core count in the range of two to 16. The trends are similar for TPC-E and TPC-C-1 and the figure omits these measurements for clarity. The legend reports in parentheses the average per distribution latencies.

A transaction's latency is the number of cycles elapsed from the moment it enters the transaction queue until it completes execution. For STREX, as *team_size* increases, the distribution tends to move toward longer transaction latencies. Figure 8 shows the corresponding relative throughput for TPC-C-10 and TPC-E demonstrating that throughput also increases with the *team_size*. With up to 20 threads per team, throughput improvements are the highest at 59% and 80% over the baseline, for TPC-C-10 and TPC-E, respectively. It would be straightforward to make the *team_size* configurable by the system, which can then set *team_size* according to its specific needs. This is similar to the request *batch_size* parameter in VoltDB, a commercial DBMS whose throughput and latency balance can be tuned [30]. Figure 7 shows that with SLICC, transaction latencies become shorter as the number of cores increases.

Table 3: *FPTable*: Instruction Footprint size per Transaction in L1-I size units.

TPC-C	`Delivery = 12, New Order = 14 Order = 11, Payment = 14, Stock = 11`
TPC-E	`Broker = 7, Customer = 9, Market = 9, Security = 5, Tr_Stat = 9, Tr_Upd = 8, Tr_Look = 8`

5.5 Combining STREX and SLICC

Data centers often vary their configuration at runtime to maximize the utilization of their resources, and to maintain QoS levels for different users. One such configuration option is changing the number of cores assigned to a particular application. Section 5.3 demonstrated that STREX performs much better than SLICC when the number of cores is not sufficient to hold the footprint of all transactions, whereas the opposite is true otherwise. Thus combining STREX and SLICC is a desirable option to enjoy the best of both worlds. A hybrid mechanism could make scheduling and migration decisions at the granularity of a code segment thus switching between SLICC and STREX depending not only on the specific number of available cores but also on the specific mix of transactions running at any moment. Such a mechanism could also take advantage of both inter- and intra-transaction overlap by using SLICC only for a portion of the transaction (e.g., the inner-loop in transactions of type `New Order` as shown in Figure 1). This work considers a simpler alternative that instead dynamically chooses between STREX and SLICC for all transactions based on the aggregate L1-I capacity available at runtime. If there is enough aggregate L1-I capacity to fit the workload's footprint, the system will decide to schedule all transactions using SLICC, otherwise, it will use STREX.

The mechanism operates as follows: (1) The system profiles the workload measuring the instruction footprint of each transaction type. The system records the measurements in a transaction footprint size table (*FPTable*). The goal is to measure the average footprint size, in L1-I size units, for all transaction types. (2) When a transaction group is ready for scheduling, the system decides, based on knowledge of the available number of cores and the contents of *FPTable*, whether to use STREX or SLICC. (3) *FPTable* updates are triggered at: system startup, workload change, new transactions types, and system reconfiguration (e.g., increasing or decreasing the available cores); all rare events.

The aforementioned system requires a mechanism to measure the instruction footprint of each transaction type. This work implements this mechanism re-using STREX's $phase_{ID}$ table while executing using SLICC. The system periodically switches to SLICC and executes a short profiling phase. In the profiling phase, a group of similar transactions is scheduled using SLICC. A random transaction is selected as a *sample* for this transaction type. The profiling phase counts the number of cache blocks touched by this thread on all cores. The profiling process works as follows. (1) All $phase_{ID}$ tables are reset to *zero* on all cores. (2) The *sample* thread is assigned with a non-zero $phase_{ID}$ value. (3) Any cache block touched by the *sample* thread is tagged with the pre-assigned $phase_{ID}$. (4) A cache block counter is incremented whenever the *sample* thread touches a block, and had to change its $phase_{ID}$ value. (5) This process repeats for different transaction types. The resulting cache block counts are rounded off to L1-I cache size units and are recorded in the *FPTable*. Once the profiling phase completes, *FPTable* holds the number of cores required by each transaction when running under SLICC. Section 5.5.1 evaluates the proposed hybrid solution.

5.5.1 Evaluation of the Hybrid Solution

Table 3 reports the footprint values recorded in *FPTable*, per transaction type during a profiling phase. For these experiments, the hybrid system updates the *FPTable* at system startup and on system reconfiguration. Thus, generating the *FPTable* once is sufficient. The profiling period is 0.2% of the overall execution time. When TPC-C and TPC-E run on more than 12 and eight cores, respectively, the hybrid solution selects SLICC, otherwise it selects STREX. Figure 6 reports the hybrid system's throughput. The hybrid system selects STREX on 2–8 cores for TPC-C, and 2–4 cores for TPC-E, and SLICC when there are more cores. Accordingly, the hybrid solution matches the best performing scheduler per core count. For TPC-E on eight cores, three transactions require nine cores. With SLICC, these transactions incur a few extra misses, however, the resulting throughput is still slightly higher than STREX.

5.6 Hardware Cost

Table 4 shows a cost breakdown of STREX's hardware components, and of the hybrid mechanism. STREX utilizes two main units: a team formation unit and a thread scheduler unit. The team formation unit is used to group similar transactions into teams. In this work STREX searches through a window of 30 threads. The *team management table* maintains information about threads until they are dispatched to a core. Each entry consists of: a unique numerical ID, a type ID, a team ID, an index within a team, and a timestamp.

The thread scheduler unit is responsible for incrementing the $phase_{ID}$ counter, tagging cache blocks with the current $phase_{ID}$ value, keeping track of the *lead* thread, monitoring instruction cache block victims, and context switching threads. The thread queue is a circular FIFO buffer. Each entry consists of a unique ID, a pointer to the thread's context in the L2 cache, and a *lead* flag bit. The size of the thread queue should be the maximum value allowed for the *team_size* configuration parameter. Most experiments set *team_size* to 10 with 20 being the maximum considered. Assuming one team management table per core, the total storage required per core by STREX is 665.5 bytes, in addition to the logic.

Table 4: Hardware Component Storage Costs.

Thread Scheduler	
Thread Queue	20-entries (12-bit ID, 48-bits pointer to thread context, 1-bit *lead* flag)
$phase_{ID}$ Counter	8-bit
Auxiliary $phase_{ID}$ Table	8-bit per cache block (512 cache blocks)
Total	5324 bits (665.5 Bytes)
Team Formation	
Team Management Table	30-entries (12-bit ID, 32-bit timestamp, 4-bit type ID, 4-bits team ID, 8-bit team index)
Total	1800 bits (225 Bytes)
Hybrid System: SLICC's Cache Monitor Unit	
Missed-Tag Queue	60-bits
Miss Shift-Vector	100-bits
Cache Signature	2K-bits
Total	2208 bits (276 Bytes)

For the hybrid system (described in Section 5.5), SLICC requires extra components that are modeled after the original article [2]. The total storage for the hybrid system is 1166.5 bytes, per core.

5.7 Replacement Policies

This section studies the interaction of STREX with several state-of-the-art replacement policies. A good replacement policy, maximizes the reuse of cache blocks. LRU performs well with applications when blocks have short re-reference periods. Applications with streaming accesses exhibit the exact opposite pattern that LRU tries to exploit and as a result tend to thrash an LRU-managed cache. OLTP workloads behave similarly to streaming applications and thus foil LRU replacement. Recent work propose replacement policies that aim at keeping part of a streaming footprint cache resident [13, 20]. Figure 9 reports the I-MPKI for eight cores, with the following replacement policies: (1) LRU, (2) LIP, and BIP [20], (3) SRRIP and BRRIP [13], and (4) STREX coupled with LRU, BIP, and BRRIP.

Figure 9 shows that for the baseline, BBRIP performs the best as it is specifically designed for streaming applications. However, STREX with LRU reduces I-MPKI more than 35% over the best replacement policy for TPC-C-10 and more than 45% for TPC-E. When STREX is combined with the alternate replacement policies little improvement results if any. The alternate replacement policies try to keep the first segment in the cache and thus quickly force evictions when execution proceeds to the second segment with STREX. This results in much more frequent context switching and thus degrades performance. Future work may further consider the interaction of replacement policies and STREX.

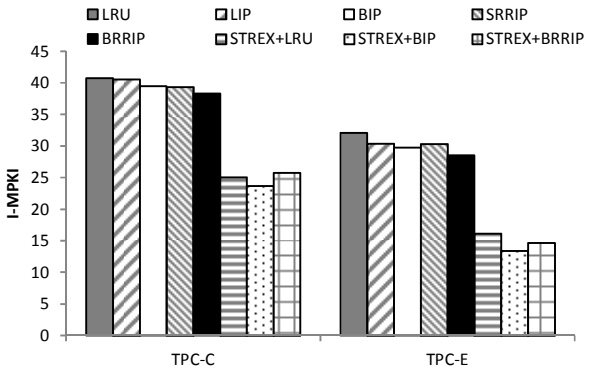

Figure 9: State-of-the-art replacement policies.

6. RELATED WORK

There have been several hardware and software proposals for reducing instruction stalls that are applicable to OLTP workloads such as instruction prefetching [7, 8, 15, 21], computation spreading [5], and transaction batching [10].

Instruction prefetching is a well-studied research area. Stream buffers [15, 21] are simple to implement in hardware, but they provide relatively low instruction coverage. More sophisticated prefetchers [7, 8] utilize book-keeping structures to record encountered instruction streams, and to replay them when part of the stream is touched again. Their structures increase area and energy. Moreover, prefetching, unless 100% accurate, increases miss traffic for fetching blocks that are never touched prior to being evicted. PIF [7] was reported to achieve near-optimal instruction coverage. Section 5.3 measured the performance of PIF. As Section 4.4 discussed, there is potential for further improving performance by combining a prefetcher with STREX.

Computational spreading demonstrated that executing the OS portion of OLTP workloads on separate cores improves instruction reuse and overall throughput [5].

STEPS [10] is a software solution whose approach is identical in spirit to STREX. STEPS relies on manual code instrumentation, which is a cumbersome task that requires high level of expertise, prone to many errors as it is manual, and results in code that is not portable since it is platform dependent. A slightly improved version, autoSTEPS, automates several components of the instrumentation process. However, the user must excite the DBMS through a client to determine where to place CTX calls manually. In addition, autoSTEPS may introduce races. STREX is a programmer-transparent technique that behaves similar to STEPS, in terms of benefits.

7. CONCLUSIONS

OLTP workloads suffer from high instruction miss stalls since their transaction instruction footprints are by far larger than current L1-I caches, on high-end server processors, thus leading to ongoing cache thrashing. This work analyzed and quantified the instruction overlap among multiple OLTP transactions of the same type. The analysis observed significant temporal instruction overlap among similar transactions, showing that their execution order can be stratified to increase instruction reuse in the caches.

This work presented STREX, a programmer-transparent, single-core technique that exploits this available temporal overlap among similar transactions. STREX groups similar transactions into *teams*, and time-multiplexes their execution on a single core, thus improving instruction and data locality. Ideally, a *lead* transaction encounters cache misses required to fetch a code segment, and its associated data, and the rest of the team hit on that code segment, and the shared data. As opposed to SLICC, which is a previously proposed technique that exploits inter- and intra-transaction locality, STREX was demonstrated to be insensitive to the available number of cores, thus outperforming SLICC when the available aggregate cache capacity is not sufficient.

Experimental evaluation demonstrated the following. When compared to a conventional CMP, STREX consistently reduces L1 instruction and data misses respectively by 37% and 13%, on average for 2 to 16 cores. When compared to an upper bound of the best instruction prefetcher known to date, PIF [6], STREX's performance is within 5% or up to 9% better, with STREX having less than 2% of PIF's storage requirements. When the core count is less than SLICC's needs, STREX outperforms SLICC by an average of 49%. This work also presented a hybrid mechanism that incorporates both STREX and SLICC, and dynamically switches to the better alternative based on the footprint size and available core count. The hybrid solution closely followed the performance of the best-performing technique.

8. ACKNOWLEDGMENTS

We thank the members of the AENAO and DIAS laboratories, and the reviewers for their comments. We thank Sudhakar Yalamanchili, Jun Wang, and the whole Georgia Tech development team for providing us with the Zesto simulator. This work was partially supported by an NSERC Discovery grant, an NSERC Discovery Accelerator Supplement, an NSERC CRD with IBM, a Sloan research fellowship, NSF grants CCR-0205544, IIS-0133686, and IIS-0713409, an ESF EurYI award, and Swiss National Foundation funds.

9. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data page layouts for relational databases on deep memory hierarchies. *VLDB J.*, 11(3):198–215, 2002.

[2] I. Atta, P. Tözün, A. Ailamaki, and A. Moshovos. Slicc: Self-assembly of instruction cache collectives for oltp workloads. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 188–198, 2012.

[3] D. Bartholomew. Qemu: a multihost, multitarget emulator. *Linux J.*, 2006(145):3–, May 2006.

[4] B. Botelho. Virtual machines per server: A viable metric for hardware selection?, 2008. Available at http://itknowledgeexchange.techtarget.com/server-farm/virtual-machines-per-server-a-viable-metric-for-hardware-selection/.

[5] K. Chakraborty, P. M. Wells, and G. S. Sohi. Computation spreading: employing hardware migration to specialize CMP cores on-the-fly. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 283–292, 2006.

[6] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2012.

[7] M. Ferdman, C. Kaynak, and B. Falsafi. Proactive instruction fetch. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 152–162, 2011.

[8] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Temporal instruction fetch streaming. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–10, 2008.

[9] M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1:117–129, 1976.

[10] S. Harizopoulos and A. Ailamaki. Improving instruction cache performance in OLTP. *ACM Transactions on Database Systems*, 31(3):887–920, Sept. 2006.

[11] IBM. WebSphere software. Available at http://www-01.ibm.com/software/websphere/.

[12] IBM. Shrinking 3900 distributed servers to 30 linux mainframes, Aug 2007.

[13] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pages 60–71, 2010.

[14] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 24–35, 2009.

[15] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, 1990.

[16] G. H. Loh, S. Subramaniam, and Y. Xie. Zesto: A cycle-level simulator for highly detailed microarchitecture exploration. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 53–64, 2009.

[17] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming language design and implementation*, pages 190–200, 2005.

[18] Oracle. Oracle WebLogic Suite 11g technical white paper. White Paper, 2009.

[19] PARSA. Data analytics benchmark with hadoop mapreduce framework, 2012.

[20] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 381–391, 2007.

[21] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 307–318, 1998.

[22] D. Sanchez and C. Kozyrakis. Flexible Architectural Support for Fine-Grain Scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[23] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. Palm: Parallel architecture-friendly latch-free modifications to b+ trees on many-core processors. *PVLDB*, 4(11):795–806, 2011.

[24] A. J. Smith. Sequential program prefetching in memory hierarchies. *Computer*, 11(12):7–21, Dec. 1978.

[25] X. Tong, J. Luo, and A. Moshovos. QTrace: An Interface for Customizable Full System Instrumentation. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 2013.

[26] P. Tözün, I. Pandis, C. Kaynak, D. Jevdjic, and A. Ailamaki. From a to e: analyzing tpc's oltp benchmarks: the obsolete, the ubiquitous, the unexplored. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 17–28, 2013.

[27] TPC. TPC benchmark C (OLTP) standard specification, revision 5.11, 2010. Available at http://www.tpc.org/tpcc.

[28] TPC. TPC benchmark E standard specification, revision 1.12.0, 2010. Available at http://www.tpc.org/tpce.

[29] VMWare Inc. Enabling end-to-end virtualization solutions for mid-market and enterprise customers: Featured case studies, marketing literature. Available at http://www.vmware.com.

[30] VoltDB Inc. VoltDB technical overview. White Paper, May 2012.